

On Balancing the Load in a Clustered Web Farm

JOEL L. WOLF AND PHILIP S. YU
IBM T. J. Watson Research Center

In this article we propose a novel, yet practical, scheme which attempts to optimally balance the load on the servers of a clustered Web farm. The goal in solving this performance problem is to achieve minimal average response time for customer requests, and thus ultimately achieve maximal customer throughput. The article decouples the overall problem into two related but distinct mathematical subproblems, one static and one dynamic. We believe this natural decoupling is one of the major contributions of our article. The *static* component algorithm determines good assignments of sites to potentially overlapping servers. These cluster assignments, which, due to overhead, cannot be changed too frequently, have a major effect on achievable response time. Additionally, these assignments must be palatable to the sites themselves. The *dynamic* component algorithm is designed to handle real-time load balancing by routing customer requests from the network dispatcher to the servers. This algorithm must react to fluctuating customer request load while respecting the assignments of sites to servers determined by the static component. The static and dynamic components both employ in various contexts the same so-called *goal setting* algorithm. This algorithm determines the theoretically optimal load on each server, given hypothetical cluster assignments and site activity. We demonstrate the effectiveness of the overall load-balancing scheme via a number of simulation experiments.

Categories and Subject Descriptors: H.3.4 [Information Systems]: Systems and Software—World Wide Web (WWW)

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Clustered Web farms, combinatorial optimization, load balancing, resource allocation problems

1. INTRODUCTION

Clustered Web farms are now becoming very popular. A key concept behind the Web farm is the notion that a number of different Web sites share pooled resources. They typically share a common front-end dispatcher to perform load control and distribute customer requests. They share the multiple Web servers themselves. And they also share back-end bandwidth to return request results to the customers. The hardware and software is typically owned, operated, and

Authors' address: IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage, and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2001 ACM 1533-5399/01/1100-0231 \$5.00

maintained by a single service provider, or content host. Examples of companies currently providing, or planning to provide, this service include AT&T, Exodus, Intel, IBM, Qwest, Verio and Worldcom.

A major challenge for clustered Web farms is to balance the load on the servers effectively, so as to minimize the average response time on the system. Overutilization of servers can cause excessive delays of customer requests. On the other hand, underutilization of servers is wasteful. Minimizing average response time will have the dual effect of maximizing the throughput the clustered Web farm can achieve. This means that the load-control algorithm in the front-end dispatcher can be made more permissive, rejecting fewer requests.

One of the main motivations for clustered Web farms is the efficiencies that arise because of the pooled resources. In this article we concentrate on efficiently pooling the server resources and on devising intelligent algorithms for the front-end dispatcher, so as to balance the load on those servers. Specifically, we do this based on the general assumption that each site is *assigned* to a cluster of one or more servers, and furthermore that these clusters may *overlap*. This means that a given server may handle customer requests from possibly multiple sites. (The alternative is partitioning the servers among the various sites, so that each server handles requests from precisely one site. Our algorithms will work for this degenerate special case as well, though less effectively.)

The rationale behind creating clusters containing multiple servers is feasibility: the distribution of customer requests among the various sites will typically be highly nonuniform. Some sites will likely be vastly more popular than others, and the popularity of a hot site will generally be great enough that assigning it to a single server would overload that server. Thus, in order to achieve acceptable performance, it will be necessary to assign some sites to multiple servers.

The rationale behind overlapping the Web site clusters is flexibility: Clustered Web farms must be able to handle customers requests from many different sites simultaneously, and this customer demand is typically bursty; see, for example, Iyengar et al. [1999] for an analysis of real Web access patterns and a corresponding analytical workload model. This burstiness can be seen on a weekly, daily, and hourly basis, due to changing site popularity and customer mix. The evidence suggests that it occurs on much smaller atomic time units as well, such as minutes and even seconds. The point here is that the average Web site access rates do not sufficiently capture the volatile nature of customer behavior. Traffic for one Web site may ebb and flow dramatically, and one site may be busy when another is less so. The hope is that a server handling several sites can be made to vary the workload balance among these sites to react to these dynamic traffic changes. Note that a partitioned cluster design does not allow this reactive capability.

See Figure 1 for an example of a hypothetical clustered Web farm. There are 3 sites and 9 servers. The network dispatcher routes customer requests for the 3 sites to the appropriate servers. The cluster for site 1 consists of servers 1–2. This overlaps with the cluster for site 2, which consists of servers 2–6. And that overlaps with the cluster for site 3, consisting of servers 6–9. The results of the customer requests are then piped back to the customer, who may then make a new request.

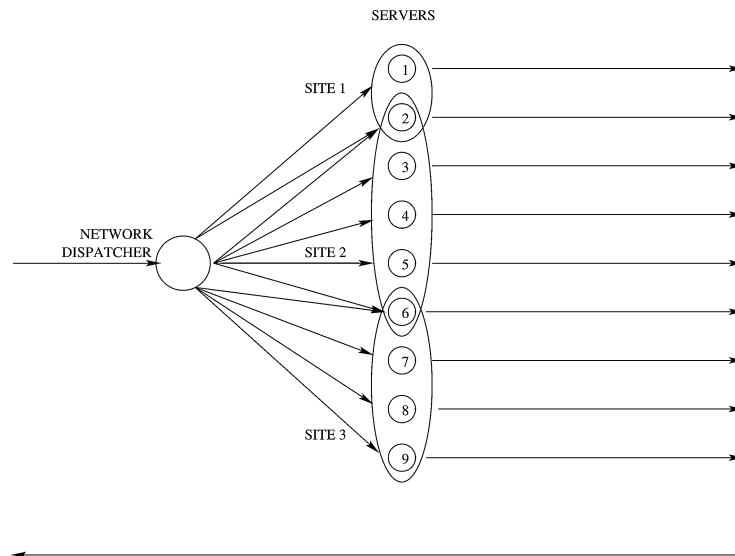


Fig. 1. Clustered Web farm.

The obvious argument for overlapping clusters in the case of Web farms is not devoid of problems, however. Although it is reasonable from a *technical* perspective that a suitably sophisticated server accommodate a modest number of Web sites, it may not be quite as reasonable from a *political* perspective. For example, an e-commerce merchant might object to sharing servers handling *transaction* requests with other merchants, even if the server happens to be *logically partitioned* among the sites [Schmunek et al. 1999]. (The objection of the merchant due to privacy grounds does not itself have to be logical.) Fortunately, such a merchant would nearly always be less sensitive to sharing servers for *browse* requests. For a load-balancing algorithm to be practical, and therefore implementable, we must deal effectively with this issue. And, indeed, we do so in this article by partitioning the workload of each site into two *categories* if necessary, one of which is not sharable and the other one which is. So we assume that the *public* requests (motivated by browsers) can share a server with sharable requests from other sites. They can also share a server with *private* requests (motivated by transactions) from the *same* site. This crucial refinement gives our algorithm the required flexibility to perform well in spite of the politically volatile issue of overlapping clusters.

Note that although the public and private categories in this article are motivated by browse and transaction requests, respectively, these terms are actually defined by the ability or inability to share. For e-commerce sites that accept the logical partitioning concept, for example, transaction requests will be counted as part of the public traffic. And, consequently, there will be no private traffic at all. At the other extreme, e-commerce sites that refuse to allow any sharing of servers, even for browsers, will have browse requests counted as part of the private traffic, and there will be no public traffic at all. And, of course, there may be noncommercial sites for which the browse and transaction terms have

no literal meaning anyway. Traffic will nonetheless be classified into either the public or private categories, depending on sharability.

In summary, we propose in this article to take advantage of the possibility of overlapping site clusters to solve the Web farm load-balancing problem very effectively. The problem decouples naturally into two subproblems, one static and one dynamic, and the solutions to each utilize the same fundamental optimization algorithm to set appropriate goals. Thus, there are three major algorithmic contributions. We outline them now.

- The key common algorithm might be called *goal-setting*. It uses an optimization technique designed to minimize the average customer response time at any given moment, given the cluster assignments of sites to servers and the current customer request load. The algorithm is a slightly special case of one used for solving the so-called *discrete class constrained separable convex resource allocation problem*, and was devised independently by Federgruen and Groenvelt [1986] and by Tantawi et al. [1988]. So this algorithm will determine the optimal load-balancing goals. Specifically, the output is the optimal number of customer requests in both the public and private categories for each site to be handled by each server in the cluster, and thus by summation the optimal number of all customer requests per server. This problem will need to be resolved on a relatively frequent basis. Fortunately, the solution technique is fast (and incremental in nature). It can handle heterogeneous servers without problems, important because some old, slower servers will inevitably be replaced by new, faster servers over time. An actually simpler and faster version can handle the special case of partitioned clusters. The goal-setting algorithm is an integral part of the static and dynamic two components below.
- The *static* component of the algorithm creates a good, we hope nearly optimal assignment of sites to servers, respecting the requirements for public and private requests. And the better this is done, the better the average response time in the goal-setting algorithm can be. The static component calls the goal-setting algorithm iteratively as it proceeds. The algorithm can be run either in *initial* or *incremental* mode. The initial mode is appropriate when configuring a new clustered Web farm. The incremental mode allows for constraints that limit the number of assignment changes, and is thus practical for maintaining high quality site-to-server assignments. A *neighborhood escape heuristic* [Garfinkel and Nemhauser 1972] is employed. The incremental mode is meant to be run periodically, perhaps once per week or so. Reconfiguring cluster assignments is obviously not a trivial task. The exact frequency will depend on the volatility of the Web site demand forecasts and the cost of doing the new assignments.
- The *dynamic* component of the algorithm performs the real-time Web server routing in the network dispatcher, based on the output of the static component and on fluctuating site customer demands. It solves a restricted subproblem based on the goal-setting algorithm to compute this idealized routing. (It does not transfer the assignment of previously dispatched requests, which could improve performance further but would presumably incur

prohibitively high overhead. We describe such an algorithm anyway because it provides an idealized bound on the quality of the dynamic component performance.) It turns out that our dynamic component scheme is a significant generalization of the standard greedy load-balancing algorithm.

Although these three algorithms might initially be regarded as mathematically complex, they are not excessively computationally expensive. (We will point out the computational complexity where it is appropriate. The speed of some of the more heuristic algorithms is governed instead by the choice of stopping condition parameters.) In fact, in all our actual experiments the running time of the algorithms was inconsequential. Given their demonstrated advantage (as we shall see) over simpler algorithms, we believe that they are worth implementing. We also point out that they are sufficiently modular—if an improved version of one of the components can be devised, the new algorithm can replace the old one.

In this article we examine the performance of our proposed Web cluster farm load-balancing scheme via simulation experiments. In particular, we show that each variant of our scheme performs better than load-balancing algorithms, which assume partitioned site to server clusters.

To our knowledge, there is no other literature on Web cluster farm load-balancing with overlapping site to server assignments. (However, see Cardellini et al. [1999] for a survey of work on load-balancing algorithms for the partitioned case.) Perhaps the lack of prior literature is due to the mathematical difficulties of the resulting optimization problems. Based on the results in this article it seems that overlapping cluster assignments are a good idea.

The remainder of this article is organized as follows: In Section 2 we present the goal-setting algorithm. In Section 3 we present the dynamic component of our algorithm. In Section 4 we present the static component. We order the sections in this way because the goal-setting algorithm is called by both the dynamic and static components. For the sake of exposition, we make one simplifying assumption in Sections 2–4, namely that requests are of uniform size. This simplifies the mathematics somewhat, but is not essential. And we list references that describe the appropriate generalizations required to handle the general case. In Section 5 we present the results of some of our simulation experiments. Section 6 contains conclusions, including some discussion of a clustered Web farm configuration-planning problem, essentially dual to the load-balancing problem which is our focus.

2. GOAL-SETTING ALGORITHM

2.1 Preliminaries

First we define some notation. (For the convenience of the reader, all key algorithmic notation in this article is summarized in Table I.) Let M denote the number of Web sites, which for convenience are always indexed by i . Let N denote the number of servers, indexed by j . Let the $\{0, 1\}$ $M \times N$ matrix $A = (a_{i,j})$ signify the potential for assignment of *public* requests for site i to server j . In other words, $a_{i,j} = 1$ if sharable requests for site i can be handled by server j ,

Table I. Notation Summary

Variable	Description
M	Number of Web sites
N	Number of servers
$A = (a_{i,j})$	Assignment matrix for public requests
$B = (b_{i,j})$	Assignment matrix for private requests
R_j	Response time function, server j
L_j	Maximum acceptable load, server j
$c_{i,j}$	Number of public requests, site i and server j
c_i	Number of public requests, site i
C	Total number of public requests
$d_{i,j}$	Number of private requests, site i and server j
d_i	Number of private requests, site i
D	Total number of private requests
$x_{i,j}$	Decision variable for optimal number of public requests, site i and server j
$y_{i,j}$	Decision variable for optimal number of private requests, site i and server j
X_j	Optimal total public load on server j
Y_j	Optimal total private load on server j
K	Neighborhood escape distance limit
T	Limit on number of assignment changes in incremental static run

and $a_{i,j} = 0$ otherwise. Analogously, let the $\{0, 1\}$ $M \times N$ matrix $B = (b_{i,j})$ signify the potential for assignment of *private* requests for site i to server j . In other words, $b_{i,j} = 1$ if nonsharable requests for site i can be handled by server j , and $b_{i,j} = 0$ otherwise. We assume in practice that a server j handling private requests for site i can also handle public requests for that site. Thus, $a_{i,j} = 1$ if $b_{i,j} = 1$. By describing the most generic case, with both public and private requests, the reader can quickly understand the algorithms in the special cases where one of the categories of requests is missing.

Associated with each server j is a function R_j measuring expected response time as a function of the customer arrival rate. This function will depend on the service time distribution, which in turn will depend on the speed of the processor. The function R_j will be increasing and convex under certain very modest conditions [Dowdy et al. 1984]. (Certainly both of these statements are intuitively plausible. Convexity, for example, simply corresponds to the law of diminishing returns.) Classic queueing algorithms exist for calculating or estimating R_j under certain simplifying assumptions about the arrival rate pattern and service time distributions [Lavenberg 1983], but in general the function may need to be evaluated via simulation experiments or monitoring. The exact formulation of this function is orthogonal to our present article. Based on this function, we can assume that there will be a maximum acceptable load L_j on server j , in the sense that exceeding this threshold will cause the value of R_j to be too large. If we set $L_j = \infty$, this maximum load constraint will, of course, be lax. Assume at a given moment that there are c_i public requests and d_i private requests in progress for site i . We break these down further into $c_{i,j}$ public requests and $d_{i,j}$ private requests for site i on server j . Thus, $c_i = \sum_{j=1}^N c_{i,j}$, and $c_{i,j} = 0$ whenever $a_{i,j} = 0$. (One cannot handle a public request for a site from

a server to which it is not assigned.) And similarly, $d_{i,j} = 0$ whenever $b_{i,j} = 0$. We let $C = \sum_{i=1}^M c_i$ denote the total number of public requests in progress, and similarly $D = \sum_{i=1}^M d_i$ denote the total number of private requests in progress.

The server loads can be regarded as optimally balanced, given the current load and site-to-server assignments when the objective function,

$$\sum_{j=1}^N R_j \left(\sum_{i=1}^M (x_{i,j} + y_{i,j}) \right), \quad (1)$$

is minimized subject to the constraints

$$\sum_{i=1}^M (x_{i,j} + y_{i,j}) \in \{0, \dots, L_j\}, \quad (2)$$

$$\sum_{j=1}^N x_{i,j} = c_i, \quad (3)$$

$$x_{i,j} = 0 \quad \text{if } a_{i,j} = 0, \quad (4)$$

$$\sum_{j=1}^N y_{i,j} = d_i, \quad (5)$$

and

$$y_{i,j} = 0 \quad \text{if } b_{i,j} = 0. \quad (6)$$

Here, $x_{i,j}$ is a decision variable representing the hypothetical number of public requests for site i which might be handled by server j . Similarly, $y_{i,j}$ is a decision variable representing the hypothetical number of private requests for site i which might be handled by server j . The objective function measures the sum of the expected response times at the various servers, which differs by a multiplicative constant from the average response time. (This constant is irrelevant from the perspective of the optimization problem.) Constraint (2) limits the acceptable load on server i . Constraint (3) ensures that the total number of site i public requests equals the actual number of such requests in progress. Constraint (4) ensures that the site-to-server assignments are respected for public requests. Constraints (5)–(6) are the corresponding requirements for private requests. If $X_j = \sum_{i=1}^M x_{i,j}$ and $Y_j = \sum_{i=1}^M y_{i,j}$ in the optimal solution, note that $X_j + Y_j$ represents the desired load on server j . Our ultimate goal will be to ensure that the optimal load $X_j + Y_j$ and the actual load $\sum_{i=1}^M (c_{i,j} + d_{i,j})$ are always close to each other for each server j .

2.2 Resource Allocation Problem

Now the optimization problem described above is a special case of the so-called *discrete class constrained separable convex resource allocation problem*. (The classes here correspond to the public and private requests for the various sites. The problem is discrete due to constraint (2); a resource allocation problem

due to constraints (3) and (5); and class constrained due to constraints (4) and (6). The separability term refers to the nature of the objective function, and the convexity term is obvious.) As shown independently in Federgruen and Groenvelt [1986] and Tantawi et al. [1988], discrete class-constrained resource allocation problems can be solved exactly and efficiently using a graph-theoretic optimization algorithm.

We now present an overview of the algorithm in Tantawi et al. [1988] as it applies to the special case above. There is a good expositional reason to do so: The graph technique of the original algorithm motivates and unifies both the dynamic and static schemes of our article.

So assuming a feasible solution exists, the algorithm proceeds in $C + D$ steps. A directed graph is created and maintained throughout the course of the algorithm. The nodes of the graph are servers $1, \dots, N$, plus a *dummy* node, which we label node 0. Set $a_{i,0} = 1$ and $b_{i,0} = 1$ for each i , and $L_0 = 0$. We create and modify a *partial* feasible solution $\{x_{i,j} | i = 1, \dots, M, j = 0, \dots, N\} \cup \{y_{i,j} | i = 1, \dots, M, j = 0, \dots, N\}$. Initially, this partial feasible solution is set for each i to have $x_{i,0} = c_i$, $y_{i,0} = d_i$, and $x_{i,j} = y_{i,j} = 0$ for all $j = 1, \dots, N$. Thus all resources reside at the dummy node. The directed graph at any step has a directed arc from a node $j_1 \in \{0, \dots, N\}$ to a node $j_2 \in \{1, \dots, N\}$ if there is at least one site i_1 satisfying

$$a_{i_1, j_1} = a_{i_1, j_2} = 1, \quad (7)$$

$$x_{i_1, j_1} > 0, \quad (8)$$

$$\sum_{i=1}^M (x_{i,j} + y_{i,j}) < L_{j_2}, \quad (9)$$

or, alternatively, at least one site i_1 satisfying

$$b_{i_1, j_1} = b_{i_1, j_2} = 1, \quad (10)$$

$$y_{i_1, j_1} > 0, \quad (11)$$

and condition (9). Condition (7) indicates that nodes j_1 and j_2 can handle public requests for site i_1 . Condition (8) indicates that a public request for site i_1 has been allocated to node j_1 . Condition (9) indicates that this request could be transferred to node j_2 without exceeding the load limit on that node. Conditions (10), (11), and (9) are the corresponding requirements for private requests. Note that there may be directed arcs *from* node 0, but there are no directed arcs *to* node 0. So the receiving node will always be a real server, not the dummy node.

The general step of the algorithm finds, among all nodes $j \in \{1, \dots, N\}$ for which there is a directed path from 0 to j , the *winning* node for which the first difference

$$R_j \left(\sum_{i=1}^M (x_{i,j} + y_{i,j} + 1) \right) - R_j \left(\sum_{i=1}^M (x_{i,j} + y_{i,j}) \right) \quad (12)$$

is minimal. (This so-called *first difference* is the discrete analog of the derivative for continuous functions, a fact which is not accidental. Also notice that the first

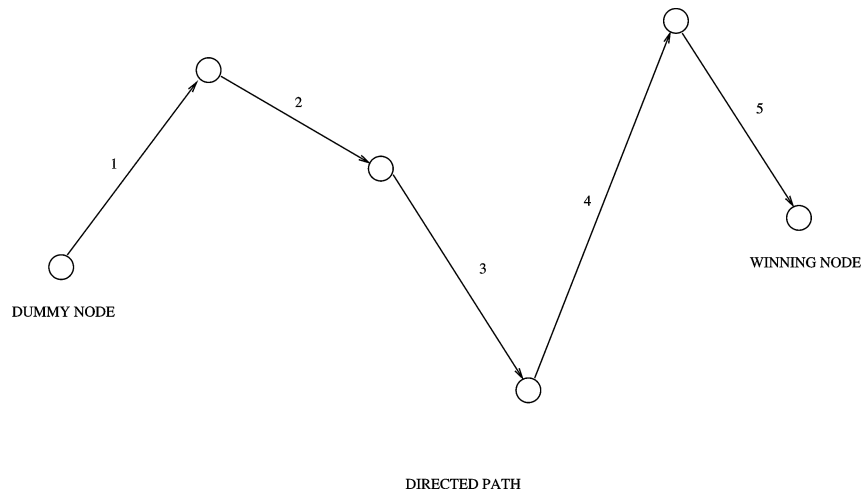


Fig. 2. Path in class constrained RAP directed graph.

differences are nondecreasing in j for each i by virtue of the convexity of R_j .) If no such node exists, the algorithm terminates with an infeasible solution. Otherwise, a shortest directed path is chosen from 0 to the winning node. For each directed arc (j_1, j_2) in this path, the value of $x_{i_1, j_1} + y_{i_1, j_1}$ is decremented by 1 and the value of $x_{i_1, j_2} + y_{i_1, j_2}$ is incremented by 1 for an appropriate site i_1 (by virtue of either a decrease and increase for a public request or a decrease and increase for a private request). Performing this step over all directed arcs has the effect of removing one unit of load from the dummy node, and adding one unit of load to the winning node. There is clearly no net effect on the load of the intermediate nodes. Thus the dummy node serves as a staging area for the resources, one of which is released in each step into the server nodes. Bookkeeping is then performed on the directed graph, which may modify some directed arcs and potentially disconnect certain nodes, and the step is repeated. After $(C + D)$ steps, the algorithm terminates with an optimal solution to the original discrete class-constrained resource allocation problem. Feasibility is guaranteed because of the conditions on the arcs in the directed graph. The complexity of this algorithm is $O(N((C + D)N + N^2 + (C + D)M))$; see Tantawi et al. [1988] for further details.

Figure 2 illustrates a path in the directed graph of the class-constrained resource allocation algorithm. Note that not all the nodes and directed arcs of the graph are shown here. Load is being transferred from the dummy node to the winning (server) node, the node connected to node 0 whose first difference is minimal. The directed path shown is intended to be the shortest such one. The directed arcs are numbered 1 through 5 for convenience. In the figure, one new (public or private) request for some site is allocated via directed arc 1, though the (server) node to which it is allocated does not change its level of activity. None of the first four server nodes have different levels of activity. The fifth winning server node experiences a gain of one unit of activity, at the expense of the dummy node.

We should observe there exist natural incremental variants of the described solution technique, so that when resolving the goal-setting optimization for a substantially similar problem instance, the running times of the algorithm can be kept quite modest. We do not go into details here.

In the special case of nonoverlapping clusters, the directed graph described above can be seen to degenerate: There will not be any directed arcs between nodes from distinct pairs of clusters, and we can assume accordingly that there are no public requests. Thus the optimization problem partitions itself into one simpler (classless) problem for each cluster. The resulting *separable convex resource allocation problem* for site i can be described as minimizing the objective function

$$\sum_{j=1}^N R_j(y_{i,j}), \quad (13)$$

subject to the constraints

$$\sum_{i=1}^M y_{i,j} \in \{0, \dots, L_j\}, \quad (14)$$

$$\sum_{j=1}^N y_{i,j} = d_i, \quad (15)$$

and

$$y_{i,j} = 0 \quad \text{if } b_{i,j} = 0. \quad (16)$$

Note that we are making the implicit assumption that all requests are private, so that the c_i and $x_{i,j}$ are all zero and the A matrix is irrelevant.

This simpler optimization problem is solvable by a fast algorithm [Fox 1966] with computational complexity $O(N_i + d_i \log N_i)$, where $N_i = |\{j \mid b_{i,j} = 1\}|$. Due to its incremental nature, this algorithm computes the optimal solution for all values between 1 and d_i as it proceeds. Thus, these values can be stored and simply looked up as needed, rather than being computed each time. (There exist even faster algorithms [Galil and Megiddo 1981; Frederickson and Johnson 1982] for this resource allocation problem, but they are not incremental in nature; see also Ibaraki and Kotoh [1988] for further details.)

As mentioned, the goal-setting algorithm can also be applied in the case of heterogeneous rather than homogeneous workloads. Details about this algorithm, which is considerably messier and less motivating than the one described above, can be found in Ibaraki and Kotoh [1988]. We omit the details here for the reasons above.

3. DYNAMIC COMPONENT

3.1 Preliminaries

In this section we describe the dynamic component of our load-balancing scheme. The algorithm assumes the (static component) assignments of sites to

servers as a given. This amounts to the determination of the matrices A and B . It then monitors exact or approximate loads at the servers, as well as the arrival of new customer requests at the network dispatcher. The job of the dynamic component is to make routing decisions in the network dispatcher for these queued requests, assigning them to appropriate servers. In doing so, it tries to achieve to the extent possible the optimal average response time and server load levels dictated by the goal-setting algorithm. Indeed, it calls the goal-setting algorithm on the traffic queued in the network dispatcher to make its decisions. The routing decisions could be implemented via an exact or a probabilistic policy, with the former obviously slightly more precise than the latter. We focus in this section on describing a *practical* variant of the dynamic component, one of which obeys the following natural restriction: Once the requests are assigned by the dynamic component to servers, they must be satisfied at those servers. In other words, the decision of the network dispatcher must be regarded as final, and requests at the servers themselves cannot be redirected based on load.

If one relaxes this restriction, one can naturally do better, at least in theory. So we also consider an *idealized* variant employing the goal-setting algorithm on the total dispatcher and server traffic. But to implement such decisions, the clustered Web farm must be able to efficiently revise previous dispatcher decisions, transferring in-progress requests from server to server appropriately. Performing such transfers with sufficiently small overhead is almost certainly too difficult a task, probably making this alternative entirely academic. Still, the idealized algorithm is useful both from an expository perspective and as a performance bound to the practical alternative.

We now describe both of these dynamic component variants, and then give an illustrative clustered Web farm example.

3.2 Practical Alternative

Let us first define some metanotation. (For the convenience of the reader, all key metanotation in this article is summarized in Table II.) At any given time, some load will be queued in the network dispatcher and some will have already been dispatched to the servers. We wish to differentiate these two types of load, as well as the overall load. So we always use a *single dot* when referring to variables that pertain to load in the dispatcher, and a *double dot* when referring to variables that pertain to load in the servers. If no dots are employed, the variables refer to the combined dispatcher and server loads.

In this practical implementation alternative, the algorithm monitors the current number of dispatched public requests $\ddot{c}_{i,j}$ and the number of dispatched private requests $\ddot{d}_{i,j}$ for site i on server j . The dynamic component does not react directly to completions of requests on the servers, but it does decrement the values of $\ddot{c}_{i,j}$ and $\ddot{d}_{i,j}$ appropriately. If this data is not readily available, it may be estimated. In the meantime, new requests arrive and queue up in the network dispatcher. Let \dot{c}_i denote the number of new public requests for site i since the last execution of the dynamic component. Similarly, let \dot{d}_i denote the number of new private requests for site i since the last execution of the dynamic

Table II. Metanotation Summary

Notation	Description
.	Dispatcher load, dynamic problem
..	Server load, dynamic problem
~	Static problem

component. Aggregating, we let $\dot{C} = \sum_{i=1}^M \dot{c}_i$ and $\dot{D} = \sum_{i=1}^M \dot{d}_i$ denote the total number of queued public and private requests, respectively, in the dispatcher.

The dynamic component may be designed to wake up and execute after a fixed time interval, after the number of items $\dot{C} + \dot{D}$ in the queue reaches some fixed batch size threshold, or perhaps some combination of both criteria. The precise details are not critical to our presentation.

We intend to apply the goal-setting algorithm to the requests queued in the network dispatcher, while leaving the previously dispatched requests alone. To do this, we require a little bookkeeping. Specifically, we define for each server j a new convex, increasing function $\dot{R}_j(z)$ by setting

$$\dot{R}_j(z) = R_j \left(z + \sum_{i=1}^M (\ddot{c}_{i,j} + \ddot{d}_{i,j}) \right). \quad (17)$$

This function simply shifts the original function to account for the amount of unperturbable load in the server. For the same reason, we also define for each server j a revised acceptable load limit \dot{L}_j by setting

$$\dot{L}_j = L_j - \sum_{i=1}^M (\ddot{c}_{i,j} + \ddot{d}_{i,j}). \quad (18)$$

With these formalities, the optimal way to dispatch the $\dot{C} + \dot{D}$ queued requests is determined by solving the following restricted goal-setting component problem: Minimize

$$\sum_{j=1}^N \dot{R}_j \left(\sum_{i=1}^M (\dot{x}_{i,j} + \dot{y}_{i,j}) \right), \quad (19)$$

subject to the constraints

$$\sum_{i=1}^M (\dot{x}_{i,j} + \dot{y}_{i,j}) \in \{0, \dots, \dot{L}_j\}, \quad (20)$$

$$\sum_{j=1}^N \dot{x}_{i,j} = \dot{c}_i, \quad (21)$$

$$\dot{x}_{i,j} = 0 \quad \text{if } a_{i,j} = 0, \quad (22)$$

$$\sum_{j=1}^N \dot{y}_{i,j} = \dot{d}_i, \quad (23)$$

and

$$\dot{y}_{i,j} = 0 \quad \text{if } b_{i,j} = 0. \quad (24)$$

Clearly, even in the optimal solution, the total load $\sum_{i=1}^M (\dot{x}_{i,j} + \dot{y}_{i,j} + \dot{c}_{i,j} + \dot{d}_{i,j})$ on server j is suboptimal overall. It is, however, optimal subject to the additional constraint that no previously assigned load can be transferred among servers.

Furthermore, note that in the special case where the algorithm wakes up whenever a new request arrives to the network dispatcher, so that either $\dot{c}_{i_1,j} = 1$ or $\dot{d}_{i_1,j} = 1$ for some site i_1 , the algorithm is simply *greedy*. In other words, if $\dot{c}_{i_1,j} = 1$, so that a new public request for site i_1 is to be assigned, the server j satisfying $a_{i_1,j} = 1$ and $\sum_{i=1}^M (\dot{c}_{i,j} + \dot{d}_{i,j}) < L_j$ whose first difference

$$R_j \left(\sum_{i=1}^M \dot{c}_{i,j} + \dot{d}_{i,j} + 1 \right) - R_j \left(\sum_{i=1}^M \dot{c}_{i,j} + \dot{d}_{i,j} \right) \quad (25)$$

is minimal is chosen. If $\dot{d}_{i_1,j} = 1$, so that a new private request for site i_1 is to be added, the server j satisfying $b_{i_1,j} = 1$ and the load-limit constraint whose first difference given by expression (25) is minimal is chosen.

The complexity of this practical dynamic component algorithm is low, since the batch size $\dot{C} + \dot{D}$ is modest. Of course, the complexity of the special case greedy alternative is trivial. It does not even explicitly require the use of the goal-setting algorithm. So our dynamic component algorithm neatly incorporates the much more traditional and simple greedy algorithm as a special case. We compare the performance of the general dynamic component algorithm with that of the special case greedy algorithm in Section 5.

3.3 Idealized Alternative

In this idealized alternative we use the goal setting algorithm to optimally assign all current load, both in the dispatcher and already at the servers, ignoring at first the possible transfers of previously dispatched load that this may entail. So we take the current number of dispatched public requests $\ddot{c}_{i,j}$ for site i on server j . Aggregating across the servers, we obtain $\ddot{c}_i = \sum_{j=1}^M \ddot{c}_{i,j}$ dispatched public requests for site i . There are also \dot{c}_i public requests queued in the dispatcher, for a total of $c_i = \dot{c}_i + \ddot{c}_i$ public requests in all. Similarly, we take the current number of dispatched private requests $\ddot{d}_{i,j}$ for site i on server j and aggregate to $\ddot{d}_i = \sum_{j=1}^M \ddot{d}_{i,j}$ dispatched private requests for site i . Given \dot{d}_i private requests queued in the dispatcher, there are a total of $d_i = \dot{d}_i + \ddot{d}_i$ private requests in all. We can thus solve the goal-setting algorithm with objective function (1) and constraints (2) through (6) to obtain the optimal loading goals $x_{i,j}$ and $y_{i,j}$. If it happens that

$$x_{i,j} \geq \ddot{c}_{i,j} \quad (26)$$

and

$$y_{i,j} \geq \ddot{d}_{i,j} \quad (27)$$

for each site i and server j , we are in good shape: For each site i , we can simply apportion the \dot{c}_i public requests and \dot{d}_i private requests at the dispatcher in

















Web Site	Cluster Servers		Partition Servers	
	Symbol	Number	Symbol	Number
circularlogic.com		3		2
realsquare.com		3		1
hexnut.com		2		2
baseball.com		2		1
triangleinequality.com		2		1
nabla.com		1		1
football.com		1		1
eggcentric.com		1		1

Fig. 3. Web site symbol glossary.

a straightforward manner to attain the optimal goals at the servers. To be precise, for site i we apportion $x_{i,j} - \check{c}_{i,j}$ of the public traffic \check{c}_i and $y_{i,j} - \check{d}_{i,j}$ of the private traffic \check{d}_i to server j . But given the likely relative cardinalities of the queued and dispatched requests, it will more likely be the case that at least one of the conditions (26) or (27) fails. Then transfers of previously dispatched load will need to be performed. Perhaps the most elegant way to envision this process is as follows: if $x_{i,j} < \check{c}_{i,j}$ for site i and server j , return a portion $\check{c}_{i,j} - x_{i,j}$ of $\check{c}_{i,j}$ to the network dispatcher. With this revised server load, condition (26) will be satisfied (as an equality). Similarly, if $y_{i,j} < \check{d}_{i,j}$, return a portion $\check{d}_{i,j} - y_{i,j}$ of $\check{d}_{i,j}$ to the network dispatcher, so that condition (27) will be satisfied. Now simply repeat the network dispatcher apportionment process described above, pushing load to the servers again. In effect, the transfers of previously dispatched load will occur during this back and forth process.

The real point is that the Web farm will not be able to accomplish such transfers without high overheads, making this idealized dynamic component alternative impractical. Its chief value here is as a bound in Section 5 to the quality of the practical alternative.

3.4 Example

The example shown in the next two figures helps illustrate the dynamic component algorithm. We consider a 15-server configuration hosting 8 sites. There are 5 servers assigned to handle public site traffic. We call these *cluster* servers, because each such server has the capability of handling traffic from multiple sites. There are also 10 servers assigned to handle private site traffic. We therefore call these *partition* servers. Each such server can handle both public and private traffic, but only from a single site. Consider Figure 3, which is a Web site symbol glossary for this example. Each symbol has a shape unique to the

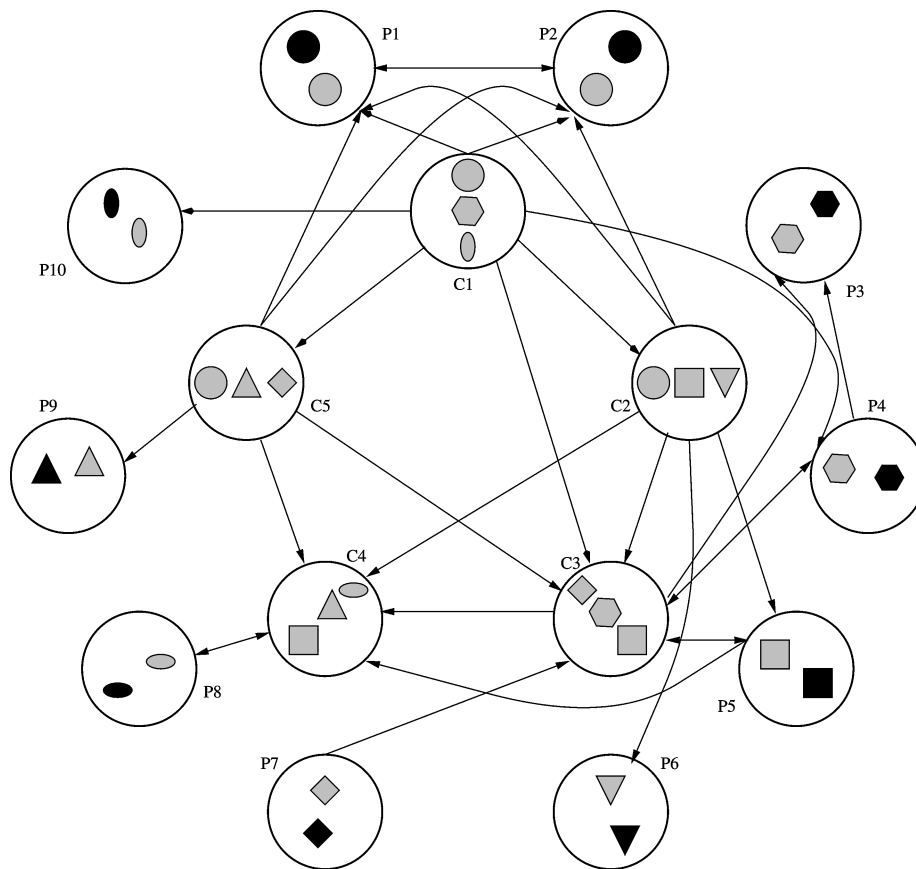


Fig. 4. The directed graph H .

site. For example, *circularlogic.com* is represented by a circle. The gray symbols indicate public traffic and the black symbols indicate private traffic. The relevant cluster and partition server cardinalities are also indicated. These are the row sums $\sum_{j=1}^N A_{i,j}$ and $\sum_{j=1}^N B_{i,j}$ of the assignment matrices A and B , respectively. For example, *circularlogic.com* is assigned 3 cluster servers and 2 partition servers.

All this is seen in more detail in Figure 4. The inner loop contains the 5 cluster servers, and the outer loop contains the 10 partition servers. Note, for instance, that *circularlogic.com* is assigned to cluster servers C1, C2, and C5. On server C1, the public traffic for this site must share the server with the public traffic for *hexnut.com* and *eggcentric.com*. Similarly, *circularlogic.com* is assigned to partition servers P1 and P2. Each of these two servers handle both public and private traffic, but only for this site.

Actually, Figure 4 conveys more information. It shows a slightly modified directed graph of the sort used by the goal-setting algorithm. Recall Figure 2, which illustrated a shortest path from the dummy node to a server node at a particular instant of time in the solution of the class-constrained resource

allocation problem. By contrast, Figure 4 shows a directed graph H showing with one exception *all* the directed arcs and nodes at such an instant. (The exception is that the dummy node and the directed arcs coming from it have been eliminated.) So H is defined as follows: The nodes correspond to the servers. For each pair j_1 and j_2 of distinct server nodes, there is a directed arc from j_1 to j_2 , provided there exists at least one site i_1 satisfying conditions (7) through (9) or conditions (10), (11), and (9).

Not all directed arcs appear in both directions in Figure 4, since they may fail to meet all the relevant constraints. For example, there is no directed arc from P3 to P4, apparently because there is no load on P3, either public (condition 8) or private (condition 11). There is no directed arc from C3 to P7, apparently because P7 is operating at full capacity (condition 9). There is no arc in either direction between C1 and P3, apparently because there is no public load on P3 (condition 8), and because C1 is operating at full capacity. There is no arc in either direction between P1 and C4, clearly because neither condition (7) nor condition (10) is satisfied.

As before, the existence of a directed arc signifies the potential for reducing the load on the server, increasing the load on another without exceeding the load capacity, and leaving the loads on other servers unaffected. A directed path from the dummy node to one of the server nodes allows the transfer one unit of load from the staging node to that server node, leaving all other server nodes unaffected. After this transfer, the graph H may be modified via a bookkeeping algorithm to respect the current status of constraints (8), (9), and (11). (The status of constraints (7) and (10) remain intact.)

One can see in Figure 4 three different types of directed arcs. One type, indicating potential transfers of either public or private load from one partition server to another, can be seen between P1 and P2. A second type, indicating potential transfers between public load from a partition server to a cluster server or vice versa, can be seen between P1 and C1. A third type, indicating potential transfers between public load from one cluster server to another, can be seen between C1 and C2.

It should be clear that the *tighter* the graph H is, in terms of having as many directed arcs as possible, the more likely it is that the goal-setting algorithm can achieve good results—yielding lower response times. This is the goal of the static component described in the next section.

4. STATIC COMPONENT

4.1 Preliminaries

In this section we describe the static component, which assigns sites to servers. The goal is to optimize the achievable performance of the goal-setting algorithm in Section 2, and therefore of the dynamic component in Section 3. The key input to the static component is forecasts of the average demand for public and private requests for each site, as well as any constraints on the allowable site-to-server assignments. The output of the static component is simply the two $\{0, 1\}$ assignment matrices $A = (a_{i,j})$ and $B = (b_{i,j})$.

Note that the issue of providing good quality forecasts is orthogonal to the main thrust of this article. So we assume these forecasts as given. It is not always possible, of course, to forecast with great accuracy. Fortunately, our static algorithm is somewhat insensitive to the exact forecasts, and the dynamic algorithm by its nature compensates for such inaccuracies.

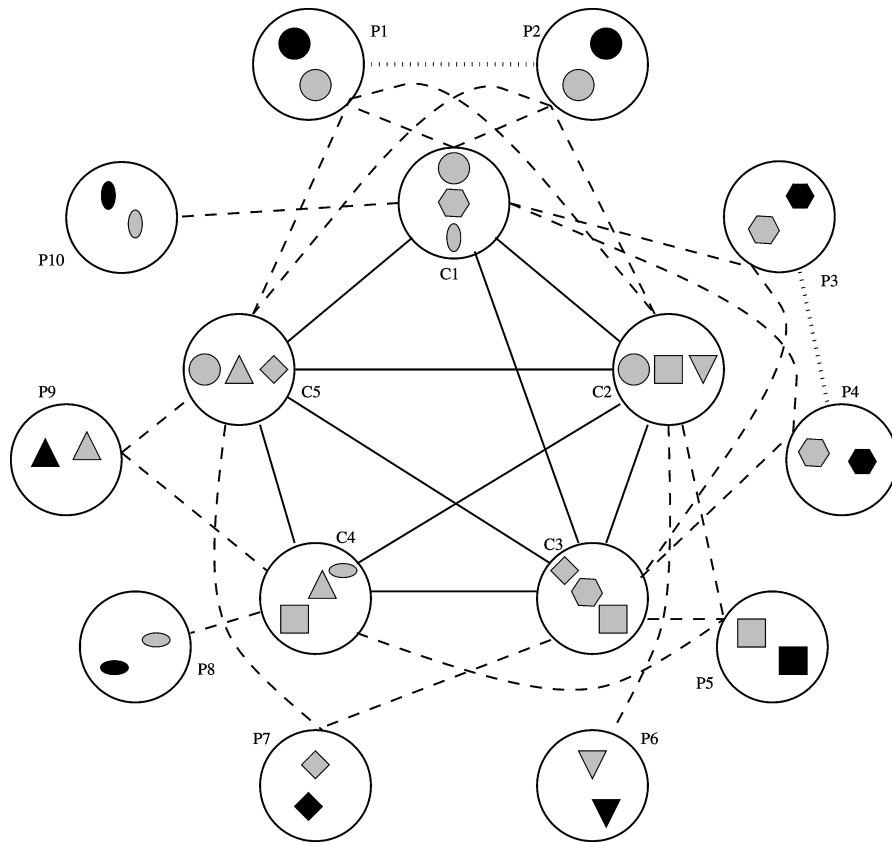
We also assume the constraints on site to server assignments as given. These constraints might pertain to the physical capacities of the servers themselves, to the goal of achieving acceptably high cache hit rates, to the operating systems on the servers, to fixed server assignments that certain sites might have negotiated, and so on. Such constraints might be quite complicated or quite elementary. For example, a cache constraint might be relatively complex, based perhaps on an analytic hit-rate model, the locality of the site data and the cache size. It would have the effect of ensuring that a relatively small number of sites be assigned to each server. Such a cache model appears, for example, in Stone et al. [1992]. On the other hand, an operating system constraint might simply be a list of sites inappropriate for certain servers. The point is that the static component is heuristic in nature and can deal with the constraints by simply checking them before considering any changes in assignments.

The static component has two possible modes. The *initial* mode is used to configure a new system from scratch, one for which no sites have yet been assigned to servers. The *incremental* mode is then used on a periodic basis to adjust existing site-to-server assignments based on revised site demand forecasts. In order to ensure that the implementation of those adjustments is practical, we allow for a constraint that limits the allowable number of site-to-server assignment changes allowed. Both modes employ essentially the same methodology. In the initial mode, one might have to employ analytic models for the response time functions, while in the incremental mode one could use more accurate measured response times instead. These presumably would better capture the effects of caching and other real-world phenomena.

The incremental static component should not be executed too frequently, since there is an obvious cost in making the assignment changes. Once or so per week is probably reasonable. The exact frequency will depend on the relative tradeoff of this cost compared with the potential for performance improvement given updated Web site forecasts. A run of the static algorithm could presumably be triggered by the detection of some sort of load imbalance condition.

The primary goal in both modes is to achieve high connectivity of the undirected graph G , defined as follows: The nodes correspond to the servers. For each pair j_1 and j_2 of distinct nodes, there is an arc between j_1 and j_2 provided there exists at least one site i_1 for which $a_{i_1,j_1} = a_{i_1,j_2} = 1$, or at least one site i_1 for which $b_{i_1,j_1} = b_{i_1,j_2} = 1$. These conditions mimic conditions (7) and (10) in the definition of the directed graph H in Section 3. Since conditions (8), (9), and (11) are generally satisfied in any well-balanced clustered Web farm, the notion is that G serves as an effective surrogate for H . Note also that conditions (7) and (10) are essentially static, while conditions (8), (9), and (11) are inherently dynamic. This is why G is more appropriate for the static component.

Figure 5 shows the graph G for the example described in Section 3. Note the similarities to the directed graph H shown in Figure 4. There is, however,

Fig. 5. The graph G .

an arc in Figure 5 between servers $C1$ and $P3$, even though Figure 4 has no comparable directed arc in either direction. And several other directed arcs in Figure 4, such as the one from server $P4$ to $P3$, are not matched by directed arcs in the opposite direction. Notice that in the figure we show arcs between two cluster servers as solid lines, arcs between two partition servers as dotted lines, and arcs between a cluster and a partition server as dashed lines.

We could attempt to increase connectivity by minimizing the *diameter* of the graph G , which is the maximum distance between any pair of nodes. Or we could attempt to increase connectivity by minimizing the *average distance* between all pairs of nodes. But both of these problems are NP-hard, and the literature on them appears to be nearly vacuous. In Wolf et al. [1997] a heuristic was proposed in a different context for the diameter minimization problem, but no guarantees were given for the solution. Indeed, we know of no approximation algorithm for either of these graph-theoretic problems. (An *approximation* algorithm [Hochbaum 1997] is a polynomial time scheme in which the objective function is guaranteed to be within a fixed multiplicative constant of optimal.)

We, therefore, adopt a slightly different approach. The objective function we attempt to optimize is that of the goal-setting component itself, which we treat

as a *black box* in our heuristic. The rationale here is that this measure, effectively the average response time, is clearly more directly appropriate than either of the other two graph-theoretic surrogates (diameter or average distance), and not significantly more computationally expensive.

The metanotation is that a tilde (\sim) refers to the static problem formulation.

We proceed by inventing a new matrix \tilde{A} , which describes the space of (in some sense *preferred*) assignments of sites to servers, as follows: $\tilde{A} = (\tilde{a}_{i,j})$ will again be a $\{0, 1\}$ matrix of size $M \times N$. Under the typical conditions that there exist both public and private traffic for each Web site, we make the translation from \tilde{A} to $A_{\tilde{A}}$ and $B_{\tilde{A}}$ in the following fashion:

$$a_{i,j} = \tilde{a}_{i,j} \quad (28)$$

and

$$b_{i,j} = \begin{cases} 1 & \text{if } \tilde{a}_{i,j} = 1 \text{ and } \tilde{a}_{i',j} = 0 \text{ for all } i' \neq i, \\ 0 & \text{otherwise.} \end{cases} \quad (29)$$

This makes the implicit assumption that *any* server j for which the row sum $\sum_{i=1}^M \tilde{a}_{i,j} = 1$ is a partition server with both public and private traffic for one site, and all other servers are cluster servers with *multiple* public-site traffic. Note that not all possible pairs of assignment matrices A and B can be regarded as translated from some matrix \tilde{A} . The ones that cannot are, however, less preferable to us. The static component will not choose them.

We have to make minor adjustments to handle special cases. If only private traffic exists for site i , we require that $\tilde{a}_{i',j} = 0$ for $i' \neq i$ whenever $\tilde{a}_{i,j} = 1$. This makes j a partition server for site i . We set $a_{i,j} = 0$ and $b_{i,j} = 1$. If only public traffic exists for site i , any server j assigned to that site can be regarded officially as public, even if $\sum_{i=1}^M \tilde{a}_{i,j} = 1$. So we set $a_{i,j} = \tilde{a}_{i,j}$ and $b_{i,j} = 0$.

We also need to modify the response time function R_j for each server j , in order to handle the cases where the load on that server exceeds its load limit L_j . Such infeasibilities may well occur, at least in the early stages of our static component algorithms, and we need to handle them by forcing feasibility at a large objective function cost. So consider a very large number Q and let $\tilde{R}_j(k) = R_j(k)$ for $k \leq L_j$, but $\tilde{R}_j(L_j + 1) = \tilde{R}_j(L_j) + Q$, $\tilde{R}_j(L_j + 2) = \tilde{R}_j(L_j + 1) + 2Q$, and so on. The general recursion defines

$$\tilde{R}_j(L_j + k) = \tilde{R}_j(L_j + k - 1) + kQ \quad (30)$$

for $k > 0$. Note that \tilde{R}_j remains convex and increasing. We also redefine the maximum acceptable load to be $\tilde{L}_j = \infty$.

Let \tilde{c}_i and \tilde{d}_i refer to the forecasted public and private load, respectively, for site i . We base our exposition on the case where both public and private traffic exists for each site, but the reader can quickly determine the appropriate modifications when this is not the case.

4.2 Initial Assignment Algorithm

Without loss of generality, assume that sites have been reindexed in terms of increasing forecasted total load $\tilde{c}_i + \tilde{d}_i$. We also assume that the servers have

been reindexed in terms of increasing performance. This could, for example, be determined by their original maximum acceptable loads L_j .

The first step of the initial assignment algorithm is the matching of sites to servers in sequence, to the extent possible. (There may be feasibility constraints that restrict certain sites from certain servers, and we must respect these.) So looping from site $i = 1$ to M in order, we set $\tilde{a}_{i,j_i} = 1$, where j is the lowest index of a feasible but currently unassigned server, and we set $\tilde{a}_{i,j}$ to be 0 for all other j . It is possible, but presumably highly unlikely, that there will not be feasible assignments possible for all sites. This might happen because of the constraints, and it might happen because $M > N$. In this case, the initial assignment algorithm has failed. Otherwise, there are M partition servers at this point, and the remaining $N - M$ servers are idle. The intuition behind the initial choice of assignments is that the lower performance servers are likely to remain partition servers during the execution of the remainder of the static component.

We evaluate the objective function for this *initial* solution, which happens to be

$$\sum_{i=1}^M \tilde{R}_j((\tilde{c}_i + \tilde{d}_i)). \quad (31)$$

The initial solution is now modified by the implementation of a so-called *neighborhood escape* heuristic. This heuristic employs a *metric* Δ to measure distances between two possible assignments of sites to servers, which we shall describe shortly. Briefly, a neighborhood escape heuristic is an iterative improvement scheme that attempts to avoid being trapped in local minima while achieving relatively low computational costs. Assuming, for the moment, the predefined metric Δ on the search space of feasible solutions, plus an existing initial feasible solution such as the one described above, the algorithm proceeds in *stages*. At the beginning of each stage there is a so-called *current* solution, which may be modified during the stage. At the beginning of the first stage the current solution is the initial solution. Each stage successively searches the neighborhoods of distance 1, 2, and so on, about the current solution. (A *neighborhood* of distance k about the current solution is the set of all feasible assignments whose distance from the current solution is less than or equal to k . A *ring* of distance k about the current solution is the set of all feasible assignments whose distance from the current solution is equal to k , which is the neighborhood of distance k minus the neighborhood of distance $k - 1$.) If an objective function improvement can be found in the neighborhood of distance 1 about the current solution, the heuristic chooses the best such improvement, relabels it as the current solution, and iterates the process by starting the next stage. If no improvement can be found within the neighborhood of distance 1, the heuristic considers the ring of distance 2 instead. Again, there are two possibilities: If there is an improvement here, the best such improvement is chosen as the current solution, and the heuristic starts the next stage. If not, the heuristic considers the ring of distance 3, and so on, up to a fixed distance limit, say K . If no objective function improvements have been reached within the K th ring, the process terminates with a final solution equal to the current solution.

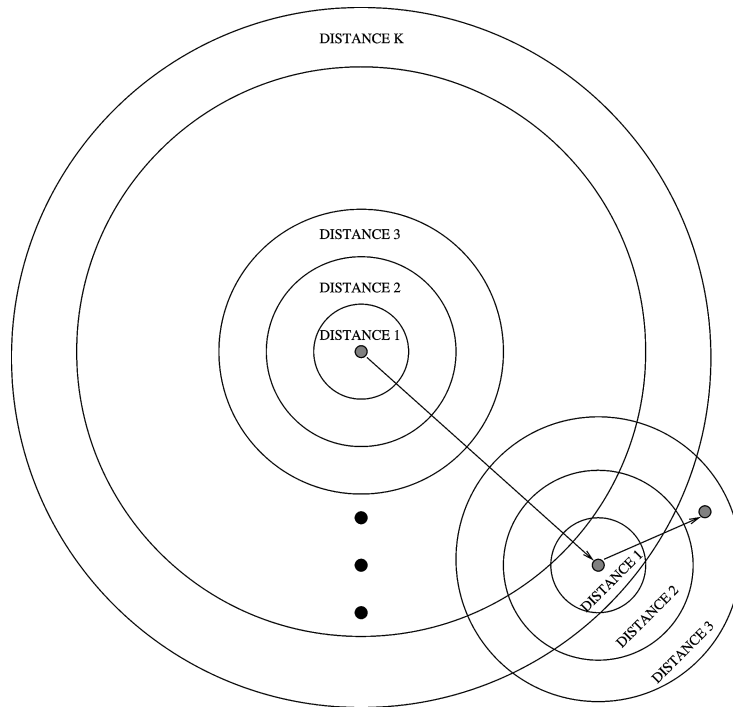


Fig. 6. The neighborhood escape heuristic.

Figure 6 illustrates this process. The current solution at the start of the initial stage is at the center of the first set of neighborhoods. The heuristic searches the first neighborhood and subsequent rings in sequence until an improvement is found. In the figure, it is only in the K th ring that an improvement occurs. Then, the heuristic resets the current solution to be the best improvement found in that search, and commences the second stage. The figure shows the neighborhoods about that solution. An improvement is now found in the 3rd ring. The third stage now commences, though we do not show it in the figure. Had the K th ring failed to yield an improvement in the first stage, the initial solution would have been the final solution.

We now define the distance metric Δ . Let $\tilde{A}^1 = (\tilde{a}_{i,j}^1)$ and $\tilde{A}^2 = (\tilde{a}_{i,j}^2)$ be two such assignments, and define

$$\Delta(\tilde{A}^1, \tilde{A}^2) = \sum_{i=1}^M \sum_{j=1}^N |\tilde{a}_{i,j}^1 - \tilde{a}_{i,j}^2|. \quad (32)$$

The intent here is simple: If, for example, we modify an assignment \tilde{A}^1 into a new assignment \tilde{A}^2 by adding or subtracting a single site to one of the servers, we obtain $\Delta(\tilde{A}^1, \tilde{A}^2) = 1$. A move of a site from an old server to a new server (which did not have that site assigned previously) has distance 2. Subtracting a site from a server and a simultaneous adding a different site to that server also has distance 2. A swap of different sites on different servers has distance 4, and so on.

For a given solution \tilde{A} and its translated assignment matrices $A_{\tilde{A}}$ and $B_{\tilde{A}}$, the test for improvement involves solving the goal-setting component problem of minimizing

$$\sum_{j=1}^N \tilde{R}_j \left(\sum_{i=1}^M (\tilde{x}_{i,j} + \tilde{y}_{i,j}) \right), \quad (33)$$

subject to the constraints

$$\sum_{j=1}^N \tilde{x}_{i,j} = \tilde{c}_i, \quad (34)$$

$$\tilde{x}_{i,j} = 0 \quad \text{if } a_{i,j} = 0, \quad (35)$$

$$\sum_{j=1}^N \tilde{y}_{i,j} = \tilde{d}_i, \quad (36)$$

and

$$\tilde{y}_{i,j} = 0 \quad \text{if } b_{i,j} = 0. \quad (37)$$

The constraints involving $\tilde{L}_j = \infty$ are vacuous, of course. But given the cost of load, which exceeds the limit L_j , we expect the solution to satisfy

$$\sum_{i=1}^M (\tilde{x}_{i,j} + \tilde{y}_{i,j}) \in \{0, \dots, L_j\}, \quad (38)$$

for each server j after very few stages in the neighborhood escape heuristic. So assignments that are feasible from all perspectives should be achievable in short order. Indeed, the first few stages of the heuristic typically assign high-volume sites to the currently unused servers, and then the heuristic will start to turn partition servers into cluster servers, typically changing the status of the higher performance servers first.

There are certainly other general-purpose combinatorial optimization heuristics that attempt to avoid the pitfall of falling into local optima. Among the more modern examples are *simulated annealing* [Laarhoven and Aarts 1987] and *tabu search* [Glover and Laguna 1997]. We could certainly have based our static component algorithms on one of these. We believe, however, that neighborhood escape heuristics [Garfinkel and Nemhauser 1972] are often less computationally expensive and yield solutions of comparable quality. They are also currently underutilized. We chose to base our algorithm on a neighborhood escape heuristic for these reasons; see, for example, a discussion of a neighborhood escape heuristic in Wolf [1989]. Obviously, a real implementation of our Web farm load-balancing algorithms could employ a competitive scheme instead.

4.3 Incremental Assignment Algorithm

The incremental algorithm is run periodically, perhaps once a week, to retain good site-to-server assignments in the presence of changing forecasts and such. Fortunately, it uses virtually the same methodology.

The initial solution is, of course, the current solution from the last static component run, and we modify it via our neighborhood-escape algorithm. The only twist is that we employ an additional stopping criterion, one which stops if the distance between the initial and current solution exceeds some user-defined threshold T . The reason is that the assignment changes are indeed expensive to implement, and we don't want to allow wildly different solutions between successive instances of static component runs.

Note that the running times of both the initial and incremental versions of the static component are governed primarily by the choices of parameters K and T . We typically employ $K = 4$, which allows for all the moves described above, and more. The value of T should depend instead on the volatility of the forecasts (providing impetus to make assignment changes) and the true costs of making them (providing impetus in the opposite direction). We do not address such questions in this article. Each objective function test involves the execution of the goal-setting component.

On the other hand, the running times for the static component are probably not all that important. Whereas the dynamic component must be carried out in the network dispatcher in real-time, the static component has the luxury of being performed on any dedicated computer. It can optimize continuously from the time the forecasts are provided to the time the assignment changes need to be implemented. This might be on the order of a day or more, far more than the amount of time required for the calculations.

5. EXPERIMENTAL RESULTS

5.1 Methodology

In this section we describe the results of simulation experiments designed to test the performance of the dynamic and static load-balancing components. We begin with a summary of the methodology.

First, we list some of the key parameters: We assumed a total of $M = 20$ Web sites and $N = 100$ servers. We also assumed that the servers are homogeneous. That is, they were all assumed to be identical in performance and capacity. In most of our experiments, we assumed that each server had enough capacity to handle 2 sites. The cluster servers therefore handled the public traffic from 2 distinct sites, while the partition servers handled the public and private traffic from a single site. There were no other assignment constraints. In one experiment we allowed each server to handle 3 sites. (The constraint of 2 or 3 sites per server might be based on the physical capacities of the servers, on cache hit rate criteria, or both.)

The algorithms in this article can certainly handle heterogeneous servers, but we do not report on such experiments here. The reasons to focus on the homogeneous case are plentiful and, we think, convincing. One advantage is that it becomes easy to gauge the quality of the resulting load-balancing. Since perfect load-balancing by the goal-setting algorithm cannot exceed that of perfectly uniform loads, that uniform load provides a bound on the quality of the solution. (Recall that the optimal solution for the goal-setting algorithm may

not yield entirely uniform load due to the site-to-server assignments and the current workload.) If we use homogeneous servers, we can visually bound the load-balancing solution quality at a glance, even though such a view may be falsely pessimistic. Better yet, if the performance happens to be close to uniform, we know we are truly doing well. A second advantage of considering identical servers is that the response time objective functions become identical, and hence irrelevant. Indeed, the goal-setting algorithm will always attempt to ship load to the reachable server that is least loaded. This eliminates the need to explicitly check the first difference formula in Eq. (12) for each server. Finally, there is no reason *not* to choose homogeneous servers in our simulation experiments. The goal-setting algorithm is, after all, exact. The optimal solution will be found whether the servers are homogeneous or not. We should therefore focus on simple cases.

We typically assumed that the arrival rates for Web sites requests were distributed according to a Zipf-like distribution. Briefly, a Zipf-like distribution [Zipf 1949; Knuth 1973] takes two parameters, M and θ , the latter corresponding to the degree of skew. The distribution is given by $p_i = n/i^{1-\theta}$ for each $i \in \{1, \dots, M\}$, where $n = 1/[\sum_{i=1}^M 1/i^{1-\theta}]$ is a normalization constant. Setting $\theta = 0$ corresponds to a pure Zipf distribution, which is highly skewed. Setting $\theta = 1$ corresponds to a uniform distribution. Setting $\theta = .5$ corresponds to a medium degree of skew. We always chose $M = 20$, of course, since that is the number of sites considered. And, in most experiments we chose $\theta = .5$. In one sensitivity experiment we chose the more heavily skewed $\theta = 0$. In most experiments we further partitioned the traffic into 90% public requests and 10% private requests. We believe such ratios are fairly typical. In one experiment, we considered the *good* case where all traffic was public, a scenario we might have if the Web sites fully accepted the logical partitioning concept. In one other experiment, we considered the *bad* case where all traffic was private, a scenario we might have if overlapping clusters were deemed politically unacceptable. Remember also that previous load-balancing literature has, to our knowledge, dealt exclusively with the partitioned cluster case. So such an experiment highlights the performance of our algorithms in that environment.

We began each simulation experiment with a call to the static assignment algorithm of Section 4. This determined good quality site-to-server assignments. We then examined each of the three dynamic schemes described in Section 3. That is, we considered the practical alternative, the greedy special case, and the idealized alternative. This last alternative may be regarded as an indication of the quality of the static component, since its real-time performance will be close to the perfect load-balancing of the goal-setting algorithm.

In order to accurately model the bursty behavior of Web traffic we followed Iyengar et al. [1999]. We actually simulated two types of customer request patterns, namely, the *log normal* and *AR(1)* interarrival processes. These two stochastic processes were seen to be appropriate in another Web-related application domain. We briefly describe both. Let Z_n denote the interrequest time for the n th (generic) request having mean λ^{-1} and standard deviation σ . The

log normal case is given by

$$Z_n = e^{\mu + \zeta \epsilon_n}, \quad n = 1, 2, \dots, \quad (39)$$

where $\mu = \log \lambda^{-1} - \zeta^2/2$, $\zeta = \sqrt{\log(\lambda\sigma^2 + 1)}$, and ϵ_n denotes the standard normal random variable with zero mean and unit variance. (In our experiments σ was chosen to be 4.0.) The *AR(1)* case is given by

$$Z_n = \lambda^{-1} + Y_n \quad (40)$$

$$Y_n = \phi Y_{n-1} + \epsilon_n, \quad n = 1, 2, \dots, \quad (41)$$

where ϕ is the autoregressive parameter of the process. (In our experiments, ϕ is 0.88.)

We report here on the log normal experiments only; but we comment that the performance of the algorithms in the AR(1) experiments is similar. Both of these arrival patterns are quite bursty, a challenging scenario for a load-balancing algorithm.

We also followed Iyengar et al. [1999] in modeling service demands via a multistage coxian distribution. We employed a 2-stage coxian with coefficient of variation equal to 2. Each site was assumed to have the same service time distribution. Again, variants of our algorithms handle the heterogeneous service time case.

In each case, the simulation experiments involved a minimum of 100,000 request arrivals and were run at the 95% confidence interval via the method of *independent replications* [Trivedi 1982]. The batching interval in the dynamic component was based on time, and designed to be equal to 1% of the average service time of a request.

5.2 Results

Figure 7 illustrates a typical set of simulation results. The figure shows the performance of the practical, greedy, and idealized dynamic component algorithms for the case of medium site skew, a capacity of 2 sites per server, with 90% public and 10% private requests.

Each curve in the figure shows the distribution of the load-balancing performance of one of the algorithms, in unit granularity. This can be interpreted as follows: For the value k on the x-axis, the y-axis of a given curve corresponds to the total percentage of times a server was within $k\%$ and $(k + 1)\%$ of uniformly-balanced load, above or below, after each execution of the dynamic component. The sum of the heights of all the y-axis values (more or less the area under the curve) is therefore 100%, and a curve heavily weighted towards the left-hand side of the figure corresponds to goodness. Again, this provides a simple but pessimistic bound on the quality of each of the dynamic component algorithms. The actual algorithms may be performing better than indicated here. We use the measure of load imbalance as the purest indication of the performance of the various algorithms. Obviously, we should be interested in response times and throughput as well; but these are secondary: The smaller the load imbalance, the better the response times and throughput will be. From the curves, it is

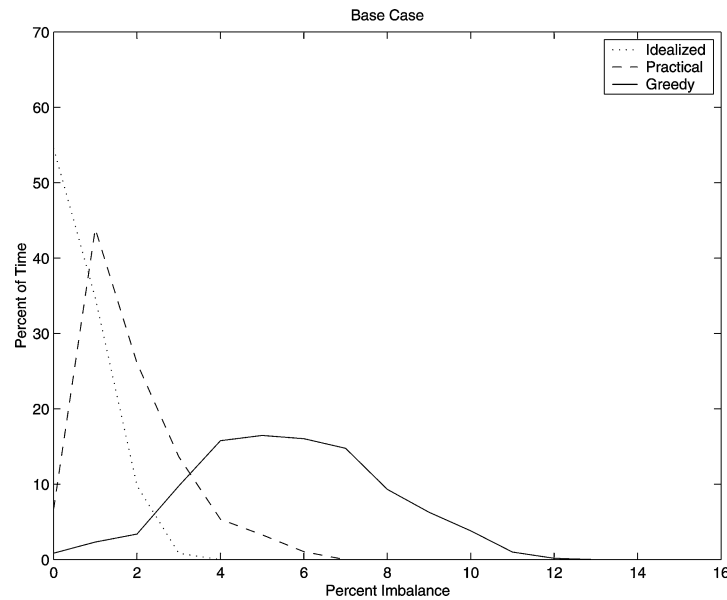


Fig. 7. Public/private traffic, 2 sites per server, medium site skew.

obvious that the greedy algorithm here does relatively less well, and that the practical algorithm has performance nearly, but not quite, equal to that of the idealized algorithm. This relative performance is observed consistently in each of our experiments.

Note how well the practical algorithm does, considering it has control of only about 1% of the requests at any given time. The idealized algorithm does better because it controls all of the requests. The greedy algorithm is clearly not that adept at load-balancing.

Since there are nearly 80 cluster servers and 2 sites per cluster server, there are roughly 160 site replications in total. Given 20 distinct sites, the average number of site replications is close to 8. Naturally, the actual numbers are skewed by the Zipf-like distribution.

We regard the results in Figure 7 as a base case, and consider the sensitivity experiments below.

Figure 8 shows the same scenario, except that all requests are now assumed to be public. Thus there are 100 cluster servers, and the average number of replications is 10. This experiment models the good case in which the Web sites accept the logical-partitioning concept. Performance is uniformly better than that of Figure 7, but the differences are not dramatic. This shows that the public/private concept is a useful one, achieving most of the benefits possible.

Figure 9 shows the same scenario again, except that all requests are now private. There are thus 100 partition servers, and load-balancing can only take place within the site partitions. The performance clearly suffers. This experiment models the bad case in which the Web sites refuse to allow overlapping clusters. Still, the practical algorithm does significantly better than the greedy one.

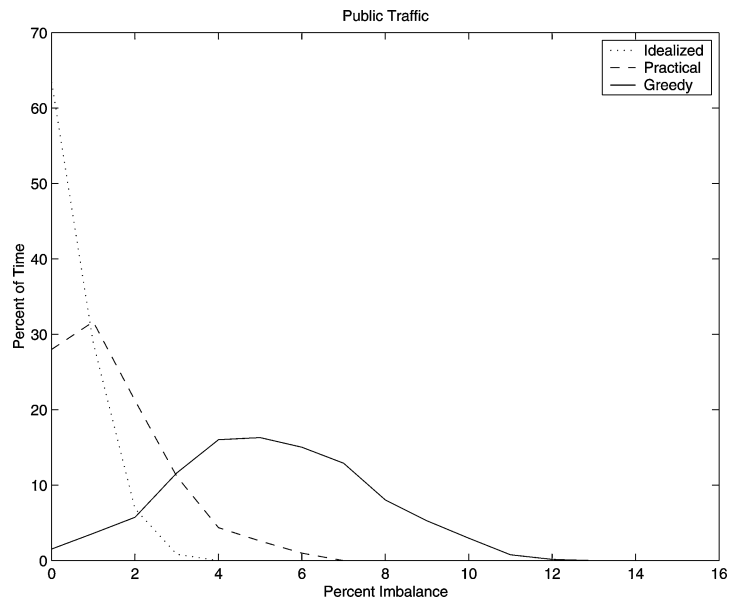


Fig. 8. Public traffic, 2 sites per server, medium site skew.

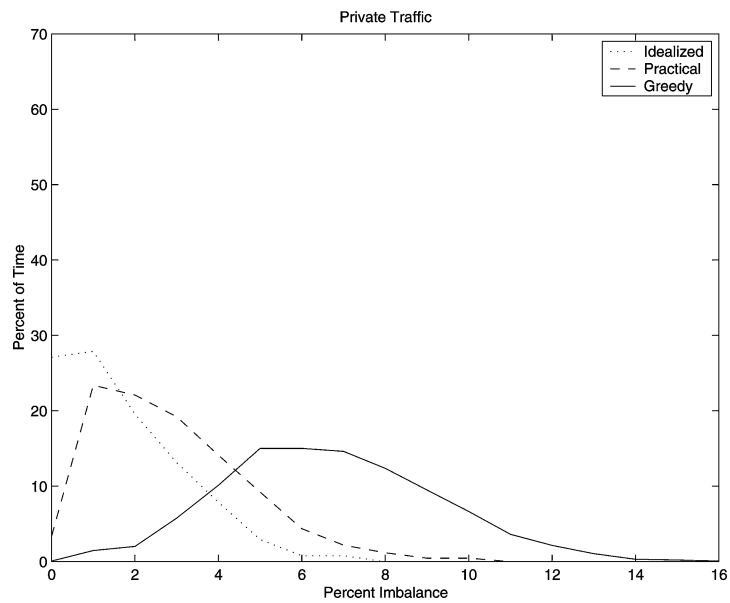


Fig. 9. Private traffic, 2 sites per server, medium site skew.

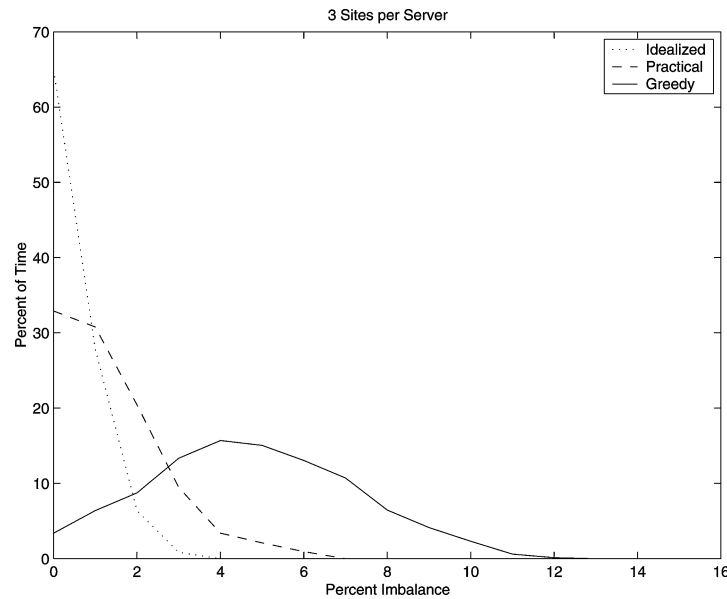


Fig. 10. Public/private traffic, 3 sites per server, medium site skew.

Figure 10 shows the case of medium site skew, a capacity of 3 sites per server, with 90% public and 10% private requests. So the difference from Figure 7 is the additional site per server. Here we attain truly excellent performance because the static component scheme is able to introduce more connectivity. Again, a little arithmetic shows that the average number of site replications is now close to 12.

In Figure 11, we consider the case of high site skew and a capacity of 2 sites per server, with 90% public and 10% private requests. So the difference from Figure 7 is the additional skew. The difference in performance is modest, though definitely visible. The degradation in performance apparently occurs because some of the partition servers for lower activity sites turn out to be underloaded.

To provide sensitivity analysis on the quality of the forecasts, we consider one experiment in which we employ the static assignment algorithm with medium skew, but employ the dynamic algorithm with high skew. This might occur, for example, if the forecasts turned out to be less skewed than the actuals. Results are shown in Figure 12. The performance is worse than either Figure 7 or Figure 11, as we would expect, but the differences are certainly not overwhelming.

We have reported here on only a small fraction of the total number of simulation experiments. It is important to reiterate that our other results are uniformly consistent with these. Naturally, the idealized algorithm always did better than the practical algorithm, but the performances of the two are not dramatically different. More importantly, the practical algorithm always performed much better than the greedy algorithm. We believe therefore that our load-balancing is quite robust, and a considerable improvement over existing algorithms.

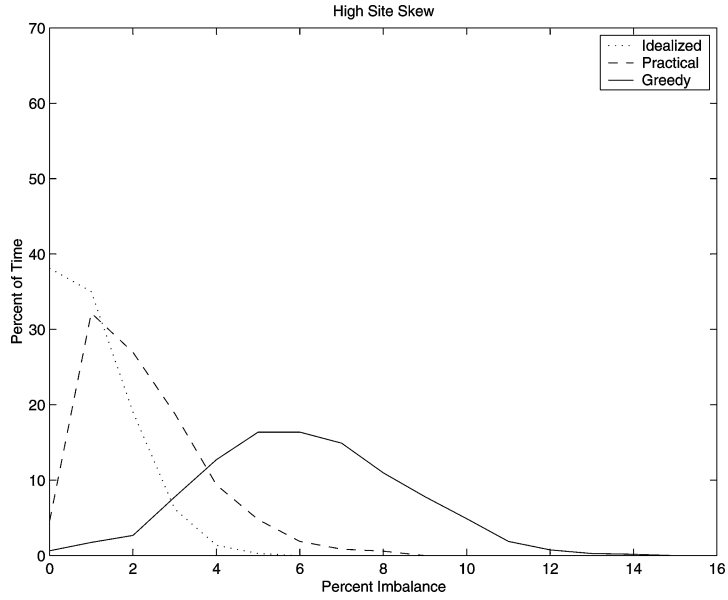


Fig. 11. Public/private traffic, 2 sites per server, high site skew.

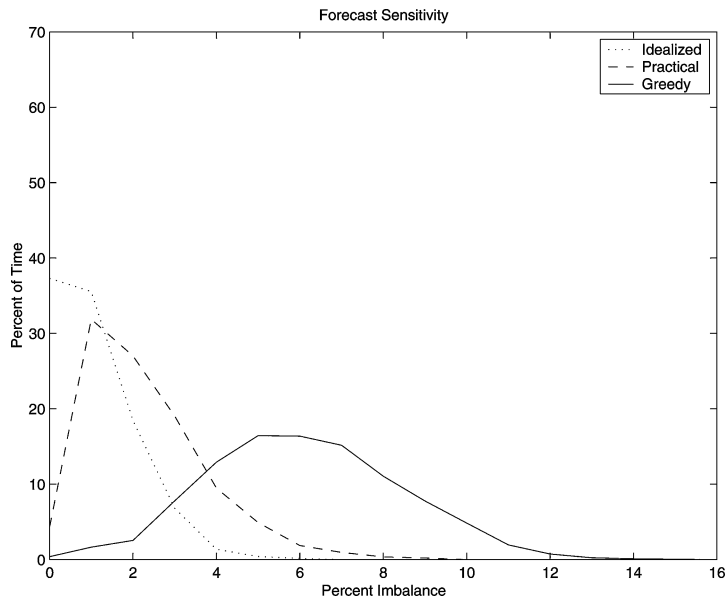


Fig. 12. Public/private traffic, 2 sites per server, imperfect forecasts.

6. CONCLUSIONS

In this article we have devised a real-time load-balancing scheme for clustered Web farms. The algorithm consists of a goal-setting scheme and a dynamic and static component. The goal-setting scheme determines the optimal load, given the assignments of sites to servers and the current load. It is graph-theoretic, and based primarily on the solution to an elaborate resource allocation problem. The dynamic component performs the actual real-time load-balancing by routing traffic in the network dispatcher to the most appropriate servers. Our scheme is a significant generalization of the more standard greedy algorithm. The static component runs on a periodic basis using forecasted loads in order to maintain good quality assignments of sites to servers. Good solutions to the static component make the dynamic component more effective. Both the dynamic and static components iteratively employ the goal-setting algorithm. The algorithms are practical in the sense that they can naturally incorporate a variety of real-world constraints, including respect for which types of site traffic can share or not share a server with other sites. The execution times of the various components are also practical.

Based on our simulation results, it seems clear that our notion of allowing overlapping clusters in Web farms is a good idea. The new notion of distinguishing public and private requests makes these overlapping clusters more palatable. Nevertheless, our load-balancing scheme works well, even when the clusters are required to be partitioned.

The problem of *configuration planning* is in a sense dual to the load-balancing problem, which has been our focus. In the latter, we wish to maximize the load we can handle in a fixed hardware configuration. In the former, we wish to minimize the cost of the configuration while handling a given site forecast demand. Thus, in principle, we can use our algorithms to solve clustered Web farm configuration-planning problems as well. Given a suite of simulation tests, we can explore the server search space to find a hardware configuration that passes the tests and has minimal cost.

Future work involves implementing our load-balancing algorithms on a prototype clustered Web farm. We will report on our experiences with this prototype in a subsequent article.

REFERENCES

- CARDELLINI, V., COLAJANNI, M., AND YU, P. 1999. Dynamic load balancing on Web-server systems. *IEEE Internet Comput.* 28–39.
- DOWDY, L., EAGER, D., GORDON, K., AND SAXTON, L. 1984. Throughput concavity and response time convexity. *Inf. Process. Lett.* 19, 209–212.
- FEDERGRUEN, A. AND GROENVELT, H. 1986. The greedy procedure for resource allocation problems: Necessary and sufficient conditions for optimality. *Oper. Res.* 34, 908–918.
- FOX, B. 1966. Discrete optimization via marginal analysis. *Manage. Sci.* 13, 210–216.
- FREDERICKSON, G. AND JOHNSON, D. 1982. The complexity of selection and ranking in $x + y$ and matrices with sorted columns. *J. Comput. Syst. Sci.* 24, 197–208.
- GALLIL, Z. AND MEGIDDO, N. 1981. A fast selection algorithm and the problem of optimum distribution of efforts. *J. ACM* 26, 58–64.
- GARFINKEL, R. AND NEMHAUSER, G. 1972. *Integer Programming*. John Wiley, New York, NY.
- GLOVER, F. AND LAGUNA, M. 1997. *Tabu Search*. Kluwer, Boston, MA.

- HOCHBAUM, D. 1997. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, Boston, MA.
- IBARAKI, T. AND KATO, N. 1988. *Resource Allocation Problems-Algorithmic Approaches*. The MIT Press, Cambridge, MA.
- IYENGAR, A., SQUILLANTE, M., AND ZHANG, L. 1999. Analysis and characterization of large-scale Web server access patterns and performance. *World Wide Web 2*, 88–100.
- KNUTH, D. 1973. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, MA.
- LAARHOVEN, E. VAN AND AARTS, E. 1987. *Simulated Annealing: Theory and Applications*. Dordrecht, Holland.
- LAVENBERG, S. 1983. *Computer Performance Modeling Handbook*. Academic Press, New York, NY.
- SCHMUNEK, G., DUPUCHE, D., FUNG, T., MYHRA, E., AND STEIN, H. 1999. Slicing the AS/400 with LPARS, IBM Redbook SG24-5439-00, Armonk, NY.
- STONE, H., WOLF, J., AND TUREK, J. 1992. Optimal partitioning of cache memory. *IEEE Trans. Comput.* 41, 1054–1068.
- TANTAWI, A., TOWSLEY, D., AND WOLF, J. 1988. Optimal allocation of multiple class resources in computer systems. In *Proceedings of the ACM SIGMETRICS Conference (May 1988)*, 253–260.
- TRIVEDI, K. 1982. *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. Prentice Hall, Englewood Cliffs, NJ.
- WOLF, J. 1989. The placement optimization program. In *Proceedings of the ACM SIGMETRICS Conference (May 1989)*, 1–10.
- WOLF, J., YU, P., AND SHACHNAI, H. 1997. Disk load balancing for video-on-demand systems. *ACM Multimedia Syst.* 5, 6, 358–370.
- ZIPF, G. 1949. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Reading, MA.

Accepted July 2001